TerraWeather
Ensuring Your Safety & Success

AI Workshop – Lecture 8

Autocaffe's Network Generator

8.0 Writing & Training Complex Networks

As you probably have experienced in the Lab 7, writing complex networks requiring several stages (“levels”) of training can be a little difficult.

The problem won't go away with Python interfaces like the ones available in Tensorflow , Keras, etc. because the network still has to be defined and loading/unloading of the trained weights needs to be performed.

In this lecture, we see how to solve these problems using Autocaffe's *network generator* (NetGen) facility. NetGen makes it much easier to write and train larger networks:

1. **Step 1**: Write your network using Autocaffe's scripting language called Mini. So, instead of `train.prototxt` and `test.prototxt`, you have `train.m` and `test.m`

2. **Step 2**: Call **`ac prep`** to prepare the results folders and run the range expansions.

3. **Step 3:** Call **`ac gen`** to convert each of your Mini scripts in the result folders into prototext files.

At this point, you can train and run your model as usual.

8.1 NetGen & Prefabs

At its core, NetGen is a translator – translating your network from Mini (which is a simple script) into Prototext format. It automatically:

1. Gives names to blobs and links, (unless you explicitly name them),
2. Connects adjacent layers automatically , for example, **`ip tanh`** would connect the `ip` and `tanh` layers.

Beyond this, NetGen uses *prefabs*, which are modules of pre-defined networks or sub-networks you can use in your own models. A rich set of prefabs is located in the **`caffe/prefab/`** directory.

8.2 An Example

The best way to learn how to use NetGen and explore prefabs is to hack simple examples. Listing 8.1 shows a model definition (train.m) for a network

consisting of multi-layer perceptrons followed by a linear combiner with 1 output:

```
\ Simple MLP network for timeseries prediction

${ $ SIZE . }   => NetworkSize
${ $ LAYERS . } => NumLayers

uses layers

: network

    "train" 873 true hdf5data train [tops] data label
    "test" 713 true hdf5data test [tops] data label

    named Dengue
        NumLayers NetworkSize 2 active mlp
        1 2 innerproduct
    end-named

    '$label loss [tops] prediction_loss

;
```

Listing 8.1 – A simple multi-layer network

Some points to note:

- Mini does not have any punctuation symbols like Python or Java. For example, in Listing 8.1, **:** and **;** are functions, not punctuation. They perform a specific task. Functions are called “words” since they need spaces just like ordinary words in English.

- Mini words are always executed from left to right, just like English. So, if you write **my3(my2(my1()))** in Python, in Mini, this would be written as **my1 my2 my3**

- A Comment (which are ignored by NetGen) is also a word **** that discards text to its right.

- The word **uses** loads all the named prefabs. For example:
  uses layers dengue
  Would load the `layers` and `dengue` prefabs.

- **network** is the “main” word that NetGen runs. So, your models should always have a `network` word. This is a user-defined word, and it is

defined using the word **:** called COLON and it is ended using the word **;** (called SEMI-COLON). These are words (ie, functions), not punctuation so they need to be spaced. For example, `:network` ... is **not** OK. `: network` is what you want. Note the space.

- The line:

```
"train" 873 true hdf5data train [tops] data label
```

creates a HDF input layer for the training phase, with named tops `data` and `label`. The ordering is important since this is what Caffe expects the HDF file to contain. The name of the training file on disk is "train.txt" the ".txt" is always implied.

- The lines:

```
named Dengue
     NumLayers NetworkSize 2 active mlp
     1 2 innerproduct
end-named
```

are the network layers. In this case, we've used the constants `NumLayers` and `NetworkSize` be determined by range expanders. A constant is defined using the word `=>` For example, `41 => hello`

- The **`mlp`** word defines a **M**ulti-**L**ayer **P**erceptron, which takes 4 arguments: the number of layers, the size of each layer, the axis on which to receive input from the bottom blob (in this case, axis=2) and the learning rate. **`active`** means a normal learning rate. **`frozen`** indicates no learning during training. These are found in the prefab called "learning", while the MLP and other layers are defined in the prefab called "layers".

- The **`innerproduct`** word defines a linear combiner. It takes two arguments, the size of the combiner's output and the bottom blob's axis to receive data on.

- Note that unlike the data layers which are un-named, the network is given a special prefix to the names, so they will be named "Dengue0", "Dengue1" , etc. This special naming stops at `end-named`

- Lastly, the (Euclidean) loss layer takes 2 inputs:

```
'$label loss [tops] prediction_loss
```

first, the last defined layer (in this case the `innerproduct` layer), and the named bottom, `'$label` which comes from the HDF data layer. The loss layer has a named top called `prediction_loss`

Quiz 8.A

You should try creating this nework and run **ac prep** and **ac gen**. Examine the output prototext it creates. Be sure to define the range expanders for the constants `NumLayers` and `NetworkSize` in your `config.txt`

Don't attempt the Lab until you complete this Quiz!

Appendix 8 contains a brief user's guide for Mini. We recommend you review Chapter 1 "**Introducing Mini**" of the guide to help you hack your prefabs.

Lab 8: Stacked Auto-encoders (SAE) using prefabs

In this lab, you will

1. Create a 3-layer autoencoder using prefabs and train it with the ae task
2. Save and use a trained autoencoder for dengue prediction

Instructions

Basic prefab layers

Each layer is defined in one line and has its own format. The parameters are written first before the layer prefab is invoked. Other prefab words may then follow which modify certain properties of that layer. Layers with fixed number of top blobs (eg InnerProduct) will have the layer tops already attached, whereas layers with flexible number of top blobs need to be configured manually using the `[tops]` word – those layers are marked with * in the table below. Note that all words after `[tops]` until the end of the line will be read as names of blobs to be added to the layer.

The formats for the layers used in this lab is given below. Parameters ending with ? are boolean values (true or false).

HDF5Data*	`<source filename> <batch_size> <load_once?> hdf5data`
• The `.txt` extension of the source filename is omitted • `load_once?` is a boolean value which selects whether the layer will load from the source file at every iteration, or just once.	
Norm	`<layer> <axis> <split?> <length> <once?> norm`
Flatten	`<layer> <axis> flatten`
Slice*	`<layer> slice`
InnerProduct	`<layer> <num_output> <axis> innerproduct`
Euclideanloss*	`<layer> <layer> loss`

In addition there are also words that build on these basic layers to provide a specific function

Bisect*	`<layer> <slice_point> <axis> bisect`
	Slices a blob into two
	`<layer> <N> truncate`

Truncate	Discards first `N` time-steps (axis 0) from blob
Select	`<layer> <index> <max_index> <axis> select`
	Extracts 1 index along an axis and junks the rest. `max_index` is the dimension of the blob axis of interest
Truncated train/test	`<layer> <N_train> <N_test> truncated-train/test`
	Discards first `N_train` time-steps during train phase and `N_test` time-steps during the test phase
SAE	`<layer> <input_length> <sequence of layer sizes> sae`
	Creates all the SAE layers needed for training
Encoders	`<layer> <sequence of layer sizes> <xt> encoders`
	Creates a set of encoders. `xt` can be "`active`" or "`frozen`"
Relabeled	`<layer> relabeled <new_name>`
	Renames the output blob of a layer to `new_name`

Utility words

train	`<layer> train`
	Apply train phase include rule to the layer
test	`<layer> test`
	Apply test phase include rule to the layer
activate	`<inner product layer> activate`
	Applies normal learning rate and weight decay multipliers
freeze	`<inner product layer> activate`
	Sets layer learning rate and weight decay multipliers to 0

"junk" blobs

Silence layers don't need to be explicitly declared in the prefab system. Just label all the unwanted blobs as "junk" and they will be added to be silenced when the prototext files are generated. Example:

```
slice [tops] junk useful-blob
```

Network definition – `train.m`

Create a `train.m` file and add the following lines

```
use layers autoencoder
```

This tells mini which prefabs to import. The autoencoder is needed for the SAE prefab.

```
: network "Dengue"

;
```

This line defines a new network called Dengue. The contents of this network is put in before the ending semicolon ; .

```
"train" 714 true hdf5data train [tops] data
 "test" 892 true hdf5data test [tops] data
```

These 2 lines reads in the data dataset from source `train.txt` and `test.txt`

```
   1 true 714 true norm
1 714 truncated-train/test
         1 flatten
```

These are the prefab equivalent of the preprocessing layers in the previous lab. Notice that the training set is truncated by 1 instead of 0 in the second line. This is a limitation of the `trancated-train/test` word that cannot accept a 0 truncation.

```
60 {{ 36 20 12 }} sae
```

This line expands to all the layers involved in training of a 3 layers autoencoder (`num_output: 36, 20` and `12` nodes in the first, second and third layers respectively), including the decoder layers and the configuration of include rules to train the SAE layer-by-layer.

Network definition – `test.m`
You can reuse the train network definition with the usual modifications – remove the train phase input layer, and use the sae-test prefab instead of sae.

```
60 {{ 36 20 12 }} sae-test
```

Generating prototext
The generation of the prototext files from the `.m` prefab files is to be done immediately after the `prep` command.

```
ac gen lab8/*
```

Check that the prototext generation was successful by making sure that the `train.prototxt` and `test.prototxt` files in the results folder are not empty. Set up the `config.txt`, `solver.prototxt`, `report.tex_template` and `metrics` files as done in the previous lab. Follow the steps in lab 7 to train the network using the ae task and generate the pdf report.

Using SAEs

After determining the SAEs that you want to use, you need to save them with the `ae-save` command

`ac ae-save lab8/3` (for example)

This will copy the caffemodel files into the weights folder (`~/caffe/weights/lab8/3` for this example. You can also save multiple models, eg `lab8/*`)

Then set up separate experiments in a new folder (eg `lab8b`) to train for the time-series prediction. Use the `metrics` and `report.tex_template` file from lab 6. The equivalent of the `train.prototxt` layers in lab 6 (without the momentum and force loss), written using prefabs, is given below – save them in a `train.m` file:

```
use layers autoencoder

: network "Dengue"

"train" 714 true hdf5data train [tops] data label
"test" 892 true hdf5data test [tops] data label

'$label 1 false 714 true norm
1 714 truncated-train/test drop
16 20 1 select

'$data 1 true 714 true norm
1 714 truncated-train/test drop

{{ 36 20 12 }} active encoders
1 1 innerproduct

loss [tops] loss
;
```

Some notes

- `'$data` and `'$label` refer to the blobs named "`data`" and "`label`"
- `drop` is required after `truncated-train/test` as `truncated-train/test` puts both train and test layers on the stack, but we only want one of them.

To make Caffe load the weights of the saved SAE, provide the path to the saved model in `config.txt`

```
AE: lab8/3
```

`test.m` is given below for reference:

```
use layers autoencoder

: network "Dengue"

"test" 892 true hdf5data test [tops] data label

'$label 1 false 714 true norm
16 20 1 select
relabeled label_16

'$data 1 true 714 true norm

{{ 36 20 12 }} active encoders
1 1 innerproduct
relabeled prediction_16
;
```

Ensure that the layers that are to be loaded (ie the inner product layers) have identical names between the saved SAE network definition (`lab8/3`) and the current experiment network definition (`lab8b`). **Otherwise the weights will not be loaded but be randomly initialized without any failure message.**

Retry the dengue prediction problem using SAEs, using the repeat task for training. The size of the SAE layers must match those used for your prediction experiment, so you will need to train and save a SAE model for each layer size and depth that you try. We suggest a depth from 3 to 5.

Post your best scores on our forum and prepare a short sharing of your results (10 mins + 10 min Q&A) at the next lecture.

Appendix 8: Mini User Guide

Mini is a simple scripting language developed at Terra Weather for data processing and to aid our AI development projects. This appendix presents a crash course in Mini in 5 chapters.

We recommend you review Chapter 1 : **Introducing Mini** to help you hack your prefabs. The other chapters are an optional read.

1 Introducing Mini

The Mini consists of:

- A ***dictionary*** which contains function definitions. Functions are called ***words***.
- Two stacks to accept input, to act as temporary storage and for output.

For example, the "Hello World" program in Mini is:

```
"Hello World" .
```

This simple program consists of two actions:

```
"Hello World"
```

puts that string on the primary stack (called the ***data stack*** or just ***stack***). The word:

```
.
```

pops the data stack and prints the result to the console.

A slightly longer program:

```
1 2 + .
```

1	pushes the number 1 on the data stack.
2	pushes the number 2 on the data stack.
+	pops the top two items off the data stack and adds them. It pushes the

	result onto the stack.
.	pops the top item off the data stack and prints it to the console.

Defining New Words

To define a new word, you use

```
:
```

and

```
;
```

For example:

```
: say-hello "Hello World" . ;
```

To run the word, you call it by its name,

```
say-hello
```

Unlike many programming languages, Mini places few restrictions to the characters that may be used for word names. For example,

```
@#$_123--
```

is a perfectly valid, (albeit unwise) function name.

Also, unlike Java, C or C++, function names are case insensitive. Eg, **`DUP`** , **`dup`** and **`Dup`** all refer to the same function/word. The preferred style in Mini is to use lowercase, except for a few exceptions like **`>R`** and **`R>`**. In this guide, we will use Uppercase initially for clarity.

Stack Manipulations

Although Mini has variables, the preferred way to program is to use the data and spare stacks exclusively. There are a number of words to accomplish this:

Word	Action
`.`	Pops the top of the stack and prints it.
`.S`	A debugging tool to non-destructively print the contents of the stack.
`DUP`	Duplicates the top of the stack. For example, **`1 DUP`** ends up with

	1 1 on the stack. Note that DUP does not re-create things on the stack; it merely duplicates pointers to things.
SWAP	swaps the top two items on the stack. For example, **"Hello" "World" SWAP** ends up with **"World" "Hello"** on the stack.
DROP	pops the top item off the stack and discards it.
>R	Pops the top of the data stack and pushes it onto the secondary stack (also known as the return stack).
R>	Pops the top of the return stack and pushes it onto the data stack.
R@	Places the top of the return stack (without popping it) an pushes the result onto the data stack.
Other useful stack manipulation words:	
OVER	**1 2 OVER** becomes **1 2 1**
NIP	**1 2 NIP** becomes **2**
TUCK	**1 2 TUCK** becomes **2 1 2**
ROT	"Rotate": **1 2 3 ROT** becomes **2 3 1**
RDROP	drops an item from the return stack.
DEPTH	Returns the number of items on the stack.

Stack manipulation words are a basic part of Mini, and you must be very comfortable using them.

Arithmetic

Mini has a separate version of mathematical operators for integers and reals:

Word	Action
+ - * / =	Converts the operands to integer values (using a floor function) and outputs and integer. Note that = is a test for equality, not variable assignment.
+. -. *. /. =.	Operates on reals & integers, outputting a real. Note that **=.** is a test for equality, not variable assignment.

> >= < <=	Works with both reals and integers.

Examples:

```
2 3 / .
0 ok

2 3 /. .
0.6666666666666666 ok

1 0.1 /
java.lang.ArithmeticException: / by zero

1 2.1 / .
0 ok

2.9 1 / .
2 ok

1 2 = .
false ok
```

Mathematical Functions

Word	Action
ceil, floor	Ceiling and floor of any number.
round	Rounds off a real.
max, min	Max/min of two numbers.
^	Exponentiation, 2 3 ^ leaves 8 on the stack.
tan, sin, cos	Trigonometric functions
#e, #pi	Constant values of e and π
exp, ln	Natural exponentiation and natural logarithms.
log10	Logarithms base 10.
to-deg, to-radians	Converts angles to degrees or radians.
mod, fmod	integer modulus and real modulus. Use **fmod** if either operand is a real number.
abs	The absolute value.
atan2	The special arctangent, very useful for computing directions from vectors. A definition may be found here:

	https://en.wikipedia.org/wiki/Atan2

Logical Operators

Word	Action
`true`	Puts the logical value “true” on the stack.
`false`	Puts the logical value “false” on the stack.
`or`	logical or
`and`	logical and

Examples:

```
true 1 2 = or .
true ok
```

Exercises

Exercise 1.1: (*)

Define **`OVER`**, **`NIP`**, **`TUCK`**, **`ROT`**, **`R@`** and **`RDROP`** in terms of **`DUP`**, **`SWAP`**, **`DROP`**, **`>R`** and **`R>`**

Exercise 1.2: (*)

Define **`FRAC`** which returns the fractional part of any number: eg: **`5.3 FRAC`** becomes **`0.3`**

2 Looping, Conditionals and Modules

Modes

The Mini runtime has 2 *modes* of operation:

- **Compilation Mode:** This mode is entered into after the word `:` (COLON) and is ended by the word `;` (SEMI-COLON). When the runtime is in compilation mode, it does not execute words, but (except for "immediate" words, which will be explained later) compiles them into the dictionary.
- **Interpretation Mode:** This mode is the default, and words entered in at the prompt or read from (eg. from a file) are executed.

In Mini, some words work in compilation mode only. For example, all conditionals (`IF...THEN`), and looping constructs may only be used in compilation mode.

Also, sometimes, we have a different compilation/interpretation version of a word, which perform the same task. For example the word `'` (called TICK) is mainly used in interpretation mode only, while its counterpart `[']` (called BRACKET-TICK) is used only in compilation mode.

Conditionals

Conditionals in Mini are represented using the words **`IF THEN`** and **`ELSE`**. These words may only be used within a word definition (ie, compilation mode). Some examples:

```
:  2?  2 = if  "yay!" . then ;
ok
3 2?
ok
2 2?
yay! ok
```

We extend this to print "nay" if the operand is not 2:

```
:  2?  2 = if  "yay!" . else "nay" . then ;
2? redefined
ok
3 2?
nay ok
2 2?
yay! ok
```

Note that unlike many languages, **THEN** ends the conditional clause. This usage follows the Forth language on which Mini is derived from.

Exercise 2.1

(*) Write a mini word **EVEN?** which prints out "yay" if the operand is even and "nay" if it is not. Make use of integer modulus, **MOD**

Multi-level Conditionals

Multi-level conditionals are a way to express more complex conditionals:

```
: goldilocks ( t -- )
     26 -
     dup 0 >  ->  "too hot!" .  |
         0 <  ->  "too cold!".  |
         otherwise  "aah! just nice" . |.
;

3 goldilocks
too cold! ok
29 goldilocks
too hot! ok
26 goldilocks
aah! just nice ok
```

Note that:

1. The tested number (t – 26) must be duplicated since both the first and second clauses consume the top of the stack.
2. `otherwise` is the same as `true ->`
3. The conditional clauses must be delimited by `|` and the final clause must be delimited by `|.`

Nested Multi-Level Conditionals

It is also possible to nest multilevel conditionals. For example:

```
: goldilocks2 ( t -- )
```

```
     26 -
     dup 0 >  ->  "too hot!" .  |
     dup 0 <  ->{   -10 < -> "brrr!" . |
                   otherwise "too cold!" . }
        otherwise  "aah! just nice" . |.
;
4 goldilocks2
brrr! ok
20 goldilocks2
too cold! ok
```

In the example:

- the word `->{` starts a nested clause,
- the word `}` terminates it.
- If the nest is on the final clause (not shown in this example), you must replace `}` with `}.`

Nested conditionals are rare, and should be avoided regardless of the language you use. However, when you really need them, Mini's nested conditionals syntax allows complex conditionals to be elegantly and legibly stated.

Exiting

You may exit a word at any time using the word `EXIT`

Eg:

```
: halfway “hello” . exit “goodbye” . ;
halfway
hello ok
```

To completely halt the Mini runtime and exit into the shell environment, use `BYE`

DO Loops

You can loop like this:

```
: counter ( n -- )
     0 do  i . loop ;
5 counter
```

```
0 1 2 3 4 ok
```

The word **DO** expects 2 items on the stack, the end and start of the loop counter. The **DO** word sets up the loop, saving the index in the return stack. For this reason, you should never amend the return stack within a **DO** ... **LOOP**.

You can also skip an index using **+LOOP**

```
: skip5 ( n -- )
     0 do  i . 5 +loop ;

10 skip5
0 5 ok
```

Nested DO...LOOPs

Unlike other programming languages, Mini's loops only allow *two* DO...LOOPs to be nested. This is not a restriction in practice, since more loops could be tucked away in other Words. An example:

```
: counter ( n -- )
     0 do
        5 0 do
          i . j . cr
        loop
     loop ;
3 counter
0 0
1 0
2 0
3 0
4 0
0 1
1 1
2 1
3 1
4 1
0 2
```

```
1 2
2 2
3 2
4 2
ok
```

DO...LOOP Gotchas

1. Only two DO...LOOP constructs may be nested. Use the words J and I to access the counter of the main and nested loop, respectively.
2. If the end < start in the DO , this causes overflow, and the loop never stops. Eg:

   ```
   -1 0 DO...LOOP
   ```
3. Never amend the return stack inside a DO...LOOP.
4. The counter words I and J use the return stack to store their state, so you can't put items on the return stack and access these variables.

Terminating a DO...LOOP

To terminate a loop use the words LEAVE and UNLOOP :

1. LEAVE simply leaves the innermost loop.
2. UNLOOP must be used if you want to exit the word.

Examples:

```
: xx 100 0 DO  i DUP . 2 = IF LEAVE THEN LOOP "Hello!" . ;
xx
0 1 2 Hello! ok


: yy
     100 0 DO
          i DUP .
          2 = IF UNLOOP EXIT THEN
     LOOP
     "I will never be printed!" . ;
yy
0 1 2 ok
```

Other Looping Constructs

The other useful Mini looping constructs are BEGIN , UNTIL and AGAIN :

Examples:

```
\ This loop will never terminate
: never-ending
          begin  “hi” . cr again ;
```

Use `UNTIL` if you need to check for a condition to terminate:

```
: 5count
     0 begin 1+  dup . dup 5 > until ;
5count
1 2 3 4 5 6 ok
```

You can also use `WHILE` loops:

```
: 4chan
     0 begin
          1+
          dup 5 < while  “Ho!” .
     repeat
;
4chan
Ho! Ho! Ho! Ho! ok
```

Explicit vs Implicit Looping

The methods of looping we've seen so far:

- Keep the state of the loop on the stacks (**`DO...LOOP`**), or expose mutable state to the programmer (eg, **`+LOOP`**),
- By definition, can't be composed and packaged as an explicit entity.

We call these looping methods *explicit looping*.

The converse of this is *implicit looping*, which does not save state on the stacks or expose looping internals in any way to programmers and which can be composed and packaged as an entity. Implicit looping is the preferred way to loop in Mini, and it uses lazy sequences and higher order functions (Chapter 3).

Use explicit looping sparingly. Use implicit looping whenever possible.

The reason for this preference for implicit looping is that:

1. Implicit loops are often easier to read and extend, and result in compact programs.
2. Implicit loops are easily parallelizable over distributed datasets, making your programs

easily extended to work over multiple computers. Explicit looping on the other hand usually makes use of mutable state (counter variables, etc.) and so, are very difficult to parallelize.

Recursion

In Mini, you can make a recursive call using the word RECURSE Eg:

```
: xx  ( n -- )
     dup .
     dup 3 > if
          drop exit
     else
          1+ recurse
     then
;
0 xx
0 1 2 3 4 ok
```

This is different from other languages, where the call would be `1+ XX` instead. However, in Mini we often use anonymous words (words with no name), so `RECURSE` is a better solution. It is also the standard way for recursion in Forth.

Exercise 2.2

2.2.1 (*) Write an iterative word (using DO...LOOP) or any other looping construct to obtain the Nth Fibonacci number.

2.2.2 (**) Write a recursive word to obtain the Nth fibonacci number.

Adding Functionality

There are 2 ways to create reusable functionality in Mini:

1. You can `include` or `import` a script. These words have the same functionality, but slightly different syntax.
2. You can create a module and use it after including/importing it.

Examples:

If you already have a file called “myscripts.m”, then you can include it:

```
include myscripts.m
```

Or, if you want to do this with a word, use import instead:

```
: xx
     “myscripts.m” import
```

```
     “myscripts2.m” import
;
xx \ at time point, both scripts would be imported.
```

Modules

So far, the words we define in Mini fall into the *default namespace.* This means that if we define a word XYZ then later want to reuse the same name for a different functionality, we lose the old functionality completely, unless we rename it. Eg:

```
\ redefine + to mean the concatenation of two strings.
: +  concat ;
+ redefined
ok
1 2 + .
12 ok
```

Which is probably not what we want. The original functionality of + (adding two numbers) has been lost. The solution is to use modules:

```
module *text
: + concat ;
end-module

\ at this stage, the default arithmetic + is enforced:
1 2 + .
3 ok
with *text
ok

\ at this point, the new textual concatenation is used.
1 2 + .
12 ok
"hello" "world" + .
helloworld ok

end-with
```

```
\ We are back to using arithmetic +
ok
45 1 + .
46 ok
```

So, the `with/end-with` words start and end the scope of applicability of a module word. This behaviour can be nested. Continuing with our previous example:

```
module *text2
: + " " swap concat concat ;
end-module

with *text2
ok
\ We now are using definitions from *TEXT2
1 2 + .
1 2 ok

\ We are now using definitions from *TEXT
with *text
ok
1 2 + .
12 ok

\ Back to *TEXT2
end-with
ok
1 2 + .
1 2 ok

\ Back to arithmetic +
end-with
ok
1 2  + .
3 ok
```

Modules may also have sub-modules (but this is rarely needed). Module names are like Mini words, but the preferred naming system is to use an initial asterisk *.

Instead of using the with keyword, it is also possible to use the module name followed by the word. Eg:

```
1 2 *text + .
12 ok
1 2 *text2 + .
1 2 ok
```

Forgetting Words

You can remove the name of a word from the dictionary, so that its functionality can't be accessed subsequently, using the word `FORGET` This is useful as part of a sandboxing solution, or to tidy up the namespace from accidental access. An example:

```
forget +
ok
1 2 + .
Error: Unknown word +
  ....
```

This only forgets words in the current/default namespace, but not in a module. For that, you need to use `FORGET` within the module itself, to remove access to intermediate words.

Extensibility

In Mini, the module system, multi-level conditionals (including **`ELSE`**), and the **`DO`**...**`LOOP`** are not "special" or baked into the language. Instead, they are actually constructed using a few "basic" Mini words. In later sections I will show you how this can be done, for Multi-Level Conditionals and **`DO`** loops. In this sense, Mini is a truly "extensible" language – much of the language can be bootstrapped from just a handful of words. These ideas are not new or even unique to Mini, but are based on its roots in Forth.

In the next section, we take our first steps towards understanding what makes Mini so powerful – we will delve deeper into its simple programming model. Mini shares much of this background with Forth.

3 Words

Introduction

This chapter covers material unfamiliar even to experienced programmers, so take your time to digest. Especially:

- Be sure to work through each and every example, first mentally then using a Mini console to confirm your understanding.
- Be sure to complete every exercise. Discuss these with your team-mates.

First-Class Functions

Mini is a language that supports *first class functions*. This means that functions can be passed as parameters to other functions. Such functionality gives Mini a lot of expressive power, and if used correctly can greatly simplify code.

For example, if you created a sorting algorithm, you have the freedom to choose to sort in ascending or descending order. In languages without first-class functions, you would be forced to either create two versions of the function (ascending-sort / descending-sort) or more realistically, let the user pass a flag to choose between these alternatives. In languages with static typing (e.g. Java), this gets much worse, since you may need to create separate versions for string arrays, integer arrays, double arrays, etc. Or, you'd have to bake-in the idea of comparing two objects into the language itself (as is the case in Java).

With first-class functions however, you would simply pass in the comparing operator (eg > , <) *as a parameter* into the single sorting function. This often results in simpler, cleaner code.

XTs

In Mini, a function is encapsulated using an object called an eXecuTable = XT. So, an XT is a record containing the following information:

1. A pointer to the code section that performs the task,
2. A pointer to the data section of a word (we'll discuss this later),
3. A flag indicating if the XT is “immediate” (we'll discuss this shortly).

You can access the XT of any word in the dictionary using the word ' (TICK) mainly in interpretation mode or ['] (BRACKET-TICK) in compilation mode

only. (NB: ' can be also used in compilation mode, but its behaviour is subtle and rarely used). Both these words are pre-fix, since they need to suppress the action of a word. You can run an XT using the word EXECUTE

Examples (recall that .S simply displays the contents of the stack):

```
' * .S
<1> com.terraweather.mini.XT@7a01fdb2
ok
>R 2 3 R> .S
<3> 2 3 com.terraweather.mini.XT@7a01fdb2
ok
execute .
6 ok
```

The dictionary is just a mapping between a string (the word's name) and an XT.

Immediate Words

When you type in a name in interpretation mode, the runtime looks up the corresponding XT and executes it. In compilation mode, the behaviour is more complex:

1. If the XT is marked *immediate* then it is executed immediately, even though we are in compilation mode. In this case, the XT is not compiled into the word being defined.
2. Otherwise, the XT's word is compiled into the current word being defined. This is the usual behaviour.

Examples:

```
\ this word is immediate:
: xx "hello" . ; immediate
ok

\ this word is not immediate, but uses XX
: yy xx 1 2 + . ;
hello ok  \ XX is executed at YY's compile time.

yy
3 ok      \ XX is not run, since it was not compiled into YY.
```

So, immediate words are created using the IMMEDIATE word, which should be placed immediately after the word definition. The use of immediate words will not be apparent at this stage, but will become clearer later.

An XT can be tested for immediacy using the IMMEDIATE? word:

```
' xx immediate? .
true ok
' if immediate? .
true ok
' dup immediate? .
false ok
```

Decompilation

It is often useful to inspect the internals of a word. There are two ways to do this: DECOMPILE decompiles a given XT to screen; SEE decompiles a named word. Examples:

(continuing from previous example):

```
see yy
[0]  Literal<1>
[1]  Literal<2>
[2]  com.terraweather.mini.number.number$8@33ea6c92
[3]  .
ok
```

(another way to accomplish exactly the same thing is to use ' YY DECOMPILE)
This decompilation clearly shows that the code body of YY does not contain any reference to XX.

The word DECOMPILE is indispensable in decompiling closures, which we will see later in this chapter.

Postpone

The word POSTPONE suppresses the immediacy of any immediate word. In Mini, POSTPONE is used so often that it has a short form, ` (BACK-TICK).

Example (continuing from our previous words XX and YY):

```
: ZZ  ` XX 1 2 * . ;
ok
zz
hello 2 ok
see zz
[0]  XX
```

```
[1]  Literal<1>
[2]  Literal<2>
[3]  com.terraweather.mini.number.number$9@6b87978e
[4]  .
ok
```

Unlike the earlier YY, ZZ now clearly calls XX. Also, the action of XX at ZZ's compile-time is suppressed by ` , which means that XX is compiled into ZZ's code body. Note that ` only affects the behaviour of XX at this one instance; XX remains an immediate word, this is not changed by `.

Switching Modes

Besides using IMMEDIATE and POSTPONE to switch immediacy, you can use the words [and] (CLOSE/OPEN BRACKET) to switch modes in any word's compile time. An example:

```
: QQ 1 2 + . [ "hello!" . cr ] ;
hello!
ok

see QQ
[0]  Literal<1>
[1]  Literal<2>
[2]  com.terraweather.mini.number.number$8@33ea6c92
[3]  .
ok

QQ
3 ok
```

The words within [...] are executed at QQ's compile time, and are not compiled into it.

Exercise 3.1

3.1.1 (*) You can see the list of words in the default namespace using the word WORDS. Find 3 words from the displayed list which are immediate.
3.1.2 (**) Write the word ` in terms of POSTPONE. (Hints: Is POSTPONE immediate? Should ` be immediate?)

The Data Section

In addition to a code section, every word has a *data section*. There is really no parallel of this in any non-Forth language. By default a word's data section is

blank, and may not be altered after it is created. The word `,` (COMMA) is used to both create and initialize a word's data section:

```
: xyz “hello” ;  41 ,
```

In this example, the data section of the word `XYZ` is set to 41. The data section is actually an array. Multiple elements can be set like so:

```
: xyz2 “hello” ; 41 , “world” ,  1 2 + ,
```

To access the data section of a word, you need the XT, and use the words `@` (FETCH) to read the value:

```
' xyz2 0 @ .
41 ok
' xyz2 1 @ .
world ok
```

or `!` (SET) to write a value:

```
777 ' xyz2 0 !
ok

' xyz2 @ .
777 ok
```

In the last line, when the index isn't specified, it is assumed to be zero. Since we will often want to “inspect” the first element of the data section, a shorthand for `@ .` is `?`

```
' xyz2 ?
777 ok
```

In practice, we will rarely assign a data sections to words constructed using the colon `:` . Instead, we often use , `@` and `!` to words created using `CREATE` and `DOES>` as we will see shortly.

Exercise 3.2

3.2.1 (**) Write a word called `ALLOT` that creates and initializes a data section for any word. The resulting data section should have length *N* and initilaized with the value 0. `ALLOT` should accept just one argument *N*. For example:

```
: say-hi “jello” . ;  6 allot
```

should create a data section for `SAY-HI` of length 6, initialized with zeros.

3.2.2 (*) Another way to create the word's data section is to do it within the word's definition like

so:

```
: say-hi2
     [ 0 , 41 , 777 , ] “yellow jello” .  ;
```

This creates a `SAY-HI2` with three elements in the data section. Note that we used the Mode Switching words `[` and `]`. Ensure that you understand & test this example thoroughly.

3.2.3 (*) What would happen if you used `ALLOT` after this like so

```
: say-hi2 .... ; 43 ALLOT
```

3.3.4 (*) Would this work?

```
: say-hi2 [ 43 ALLOT ] .... ;
```

3.2.5 (*) Define a word `::` that can be used to simultaneously define new words and create their data section like so:

```
32 :: xx “hello!” . ;
```

Would create the word `XX` *and* initialize its data section to be a 32 element array.

Delaying Decisions

Often, you know how to use a word `X` within word `Y`, but want to defer defining `X` for later, or maybe want to use a variety of versions of `X`: `X1`, `X2`...

1. The best way to do this is to use first-class functions idea – pass the XT into the word using the stack. The sorting example at the start of this chapter is an example. We will discuss this method in later chapters extensively.

2. The alternative is to define a global word which has no real code body, then bind that name to an XT at runtime. It's this latter usage that we examine here.

To create the “blank” word, use **`DEFER`**. For example, to create a blank word `XX`

```
defer xx
```

You can use this “word” `XX` anywhere subsequently. To put in some functionality, you'd use **`AS`**

```
: x2 "pug!" . ;

' x2 as xx

xx
pug! ok
```

You may change the behaviour of XX at any time using AS.

Extending Mini

In Chapter 2, I hinted that Mini is an *extensible* language, meaning you can (and are encouraged) to extend the language depending on the work at hand. For example, many of the important control words in Mini like DO, LOOP, ELSE, POSTPONE and all the multi-level conditional words have been defined as language extensions.

The first step to extending Mini is to understand the words CREATE and DOES>:

Word	Action
CREATE	Creates a new dictionary entry, with the name given at runtime. In other words, a name – XT binding is added to the dictionary. 1. The XT created this way has a special code section – it just pushes the XT to the stack. 2. The data section can optionally be defined using the COMMA (**,**) operator.
DOES>	Optionally assigns further functionality to the dictionary entry from CREATE. The functionality is delimited/ended by **;**

Examples:

We create a variable MYVAR, which is initialized to 3:

```
create myvar 3 ,
ok
myvar .
com.terraweather.mini.XT@58a07808 ok
myvar ?
3 ok
```

Be sure you understand this example thoroughly before proceeding. Note that:

1. CREATE defined a new word MYVAR on the dictionary; When it is run, MYVAR puts its XT on the stack.
2. After CREATE defines the new word, the comma operator initializes its data section with a single element, whose value is 3.

So, in this example, we have created one global variable.

In the next example, we create a "keyword" VARIABLE that when run, creates and initializes a global variable:

```
: VARIABLE ( n -- )
     CREATE , ;

32 VARIABLE HELLO
ok
HELLO ?
32 ok
41 HELLO !
ok
HELLO ?
41 ok
```

Again, be sure you understand this example thoroughly before proceeding. This is closer to the typical usage of CREATE, which is used to construct new keywords.

Exercise 3.3

3.3.1 (*) Can VARIABLE be used within a word to declare new variables?
3.3.2 (*) How does CREATE differ from the colon operator : ?

Next we see how you can extend the simple XT constructed by CREATE using DOES>.

Examples:
In the following example, we create a constant THREE that is permanently assigned the value 3.

```
create three 3 , does> @ ;
ok
three .
3 ok
```

In this example, DOES> extends the XT created by CREATE. Here's what happens:

1. CREATE created an XT whose code section contains one instruction: *Push the XT onto the stack.* (this instruction is called *PushXT*, and should be visible on a decompilation)
2. When DOES> runs, it switches over to compilation mode. The subsequent words are then compiled into the code section of the XT. So, the first instruction is still *PushXT*, but the subsequent ones are the words past DOES>. In this case, just the additional @ is compiled into the code section of the XT.
3. The word ; switches back to interpretation mode.

Just like VARIABLE, we can package this into a new "keyword" called CONSTANT:

```
: CONSTANT ( n -- )
  CREATE , DOES> @ ;

41 constant hello
ok
hello .
41 ok
```

In this case:

1. DOES> is an immediate word, so it runs even though we are in compilation mode.
2. The action of DOES> is to complete the enclosing word CONSTANT, and begin a new anonymous/nameless word. This anonymous word is the one whose functionality is added to the XT's code section as before.

Exercise 3.4

3.4.1 (*) Decompile CONSTANT using SEE. How many instructions are there? Explain what each instruction does.

3.4.2 (*) What happens if we do THREE ? .Explain your observations.

3.4.3 (*) Is it possible to amend the value of THREE? If "yes" write code to do this.

3.4.4 (*) Write a "keyword" VECTOR2 that creates a global variable representing a dimension 2 vector.

3.4.5 (**) Write a "keyword" VECTOR that creates an N dimensional vector variable. N should be an input into VECTOR. Write also a "keyword" DIMENSION that given a vector, puts its dimension on the stack.

Semantics

As we've seen, words can have different behaviour depending on whether we are in compilation mode or interpretation mode. The technical term for "behaviour" is **semantics**.

1. The *default* interpretation semantics of a word is to run its code section.
2. The *default* compilation semantics of a word is to append its code section to the enclosing word being defined.
3. An immediate word breaks the default compilation semantics, so that the word's code section is run instead of being compiled.
4. Some words may break both default semantics. Some words have no interpretation semantics (example?)

Compiling Code Programmatically

In some situations, you will have to compile code into an XT programmatically, without expecting it to have the default compilation semantics.

With these words, you can write code that writes code. There are 2 words to help you with this:

Word	Action
[COMPILE]	Takes the following word and compiles its code section into the enclosing word, *regardless of that word's compilation semantics.*
COMPILE,	Takes a given XT and compiles its code section into the enclosing word.

Examples:

```
: hi "Hi!" . ;
: bb [compile] hi "bye!" . ;

bb
Hi! bye! ok
ok

see bb
[0]  HI
[1]  Literal<bye!>
[2]  .
ok
```

Exercise 3.5

3.5.1 (*) Try out the preceding Example. How would it change if you used `POSTPONE` instead of `[COMPILE]` ? Explain and check your answer using `SEE`.

3.5.2 (*) In this Example, what are the compilation semantics of `HI`? Is using `[COMPILE]` or `POSTPONE` of any value? Explain and check your answer using `SEE`.

3.5.3 (*) If `HI` were immediate, what would your answer to 2.5.2 be? Explain and check your answer using `SEE`.

Similar to `[COMPILE]` , `COMPILE,` (note the comma!) compiles a given XT on the stack to the current word.

Examples:

```
: aa [ ' dup compile, ] ;
```

```
see aa
[0]  com.terraweather.mini.core.core$2@70a2470c
ok
```

Exercise 3.6

3.6.1 (*) Try out the preceding Example. What does `AA` do? Why did we have to use the switch mode operators?

3.6.2 (**) Write `[COMPILE]` in terms of `COMPILE,` and other words you know, except for `POSTPONE` or `` ` ``.

The use of these compilation words will be more apparent later when we consider macros in Chapter 5.

Literal Values

In the light of the preceding discussion, Numbers and Strings are not treated differently from words in Mini. Both also have compilation and interpretation semantics:

- The interpretation semantics of numbers and strings is to put them onto the stack.
- The compilation semantics of these is to create a special XT containing the number/string and compile this into the enclosing word. The XT is one which when run puts the number/string onto the stack.

Example:

```
: hi "hi" . ;
see hi
[0]  Literal<hi>
[1]  .
ok
```

This example shows that the string “hi” is contained by a word called Literal. You can control the compilation semantics of an non-word using the word LITERAL. LITERAL takes whatever's on the stack (at compile time) and compiles it as a literal into the enclosing word. There are 2 contexts in which this may come in useful:

1. There will be many cases where you will create new objects (in Java for example) and want to embed them as literals in your Mini functions. Since these are custom objects, Mini can't determine what their

compilation semantics is. In this case, you have to use the word `LITERAL` to ensure they are properly compiled into the word. I will defer an example to Chapter 8, when we cover the Mini-Java interface.

2. There are sometimes situations where you want the word definition to depend on data present at compilation time. An example follows:

Example:

```
: xx [ 1 2 + literal ] . ;
ok
xx
3 ok
```

Exercise 3.7

3.7.1 (*) What would happen in the preceding example if we omitted `LITERAL` ? Explain and confirm your answer with the Mini console.

3.7.2 (**) Write `[ ' ]` in terms of `LITERAL`, `` ` `` and `' `. Is `[ ' ]` immediate?

Parsing and the Token Stream

A line of Mini code or a Mini script in a text file is represented as a *token stream* – a sequence of strings delimited by spaces. This token stream is then interpreted token by token. Each token represents a word or number or string. Quite often your word needs to access tokens ahead of it in the token stream. You can do this using the `PARSE` word:

Word	Action
PARSE	`PARSE` takes one argument, a *character* telling the word when to stop
[CHAR]	converts the following token into a character. Some useful characters are \n (carriage return) \r (linefeed) and \t (tab)
BL	a character representing a space.

Example:
Suppose you want to read the entire line to the carriage return, and put the result on the stack:

```
: chomp [char] \n parse ;

chomp Hello World!
```

```
.
Hello World!
```

The line comment word `\` and the stack comment word `(` are similarly defined:

```
: \ [char] \n parse drop ; immediate
: ( [char] ) parse drop ; immediate
```

Local Variables

In Mini, the use of global variable is discouraged; This is actually good programming practice in any language. Global variables foster tight coupling between your words, making “action at a distance” effects that can be hard to debug.

Mini does have *local variables* which are temporary bindings valid only within a word, and these are used extensively. However, you are strongly encouraged to use these only if the stack manipulation words are clumsy. Often, a simple refactoring removes the need for local variables and keeps your code simple. This is often also faster compared to using local variables.

As a rule of thumb, if you should not use more than 3 arguments to any word. If you use more, then it is likely that you should refactor the code.

In Mini, local variables are indicated using the word `{` which is delimited by `}`. The latter is not a word, just a textual delimiter. The variables are bound to items on the data stack:

Example:

```
: add { a b }
     a b + { c }
     c . ;
1 2 add
3 ok
```

Note that the bindings can be made anywhere in the word, and you can also shadow the names too (ie, replace `c` with `a` or `b` in this example).

Think twice about using local variables. They are often not needed!

Anonymous Words

A central concept in Mini (very different from its parent language, Forth) is the use of anonymous words. These are also known as *anonymous functions* or *lambdas* in other languages. In Mini, an anonymous word is just an XT with no name:

Word	Action
`:> ... ;`	The words `:>` and `;` create an anonymous word in interpretation mode. This results in an XT on the stack.
`[: ... ;]`	The words `[: ... ;]` are applicable only in compilation mode, and form what is known as a *quotation.* When the enclosing word is run, they create an anonymous word. Quotations can be nested, and we will examine them extensively in Chapter 5 (Macros).
`EXECUTE`	this word runs *any* XT on the stack, whether anonymous or retrieved using `'`

Examples:

```
:> "I have no name!" . ;
ok
.S
<1> com.terraweather.mini.XT@560d120a
ok
execute
I have no name! ok

: make-greeting ( "greeting" -- xt )
     { g } [: g . ;] ;


"Hi Arnold!" make-greeting .S
<1> com.terraweather.mini.XT@3f78807
ok
execute
Hi Arnold! ok
```

In the second example with the quotation, we used a local variable to bind the item on the stack and to call it within the quotations. There is no other way to do this except using local variables. You can decompile the resulting XT to get a deeper feel of what's going on:

```
"Selamat Pagi Chris!" make-greeting decompile
[0]  Literal<Selamat Pagi Chris!>
[1]  .
ok
```

You can see here that for quotations, the local variable has been inlined as a literal into the XT.

Function Bindings

Since we use first-class functions frequently in Mini programming, it would be helpful to let local variables create *local functions* (ie, which run instead of being put on the stack like a local variable). There are two ways to do this in Mini:

1. Bind the XT to a local variable as usual, then use `EXECUTE` to run it as needed,
2. Use the ~ (TILDE) word to convert the XT to a local function instead, so that the word is run when it is called. This is the preferred method.

Continuing from the previous example:

```
"Hello Sam!" make-greeting dup .S
<2> com.terraweather.mini.XT@.. com.terraweather.mini.XT@...

: how-are { xt } "I would like to say " .  xt execute ;

how-are
I would like to say  Hello Sam! ok

: how-are ~ { fn } "I say:" . fn ;

how-are
I say: Hello Sam! ok
```

Exercise 3.8

3.8.1 (*) Test out these examples for yourself.

3.8.2 (*) Write a version of how-are that does not use either local variables or local functions.

Exception Handling

Mini provides a number of words for exception handling. The following table describes these words. (The Stack Changes are before/after changes to the stack, and z*x = x1,x2...xZ , ie, Z items on the stack.)

Word	Action
`THROW`	This word halts the further execution of a word, and puts execution at the point CATCH was last called. THROW takes one argument, the error number. Stack Changes: (n = error number)

	`k*x n -- k*x n`
CATCH	This word takes an XT and runs it. If somewhere in the calling hierarchy the system THROWs an exception, the XTs execution is halted, and the THROW's error number is placed on the stack. In the case of an error, the depth of the stack is left unchanged. Stack Changes: (0 = no error , n = error number) `i*x xt -- j*x 0 | i*x n`
TRY	Executes the given XT, signalling an error if thrown. Two kinds of errors are caught: 1. Java errors are signalled using the error number -3. Also the java error object itself is put on the stack. 2. Mini exceptions using the THROW mechanism is signalled using the error number -2. Stack Change: (n=-3 for a java error, n=-2 for a THROW generated exception) **`i*x xt -- j*x false | k*x error n true | l*x n true`**
ABORT	Unconditionally sets the runtime's exception flag and creates a multi-level exit.
EXCEPTION?	Puts the runtime's exception flag on the stack. Stack Change: `-- f`
JAVA-ERROR?	Determines if the object on the stack is a java exception. Stack Change: `x – f`

Examples:

The examples shown will be for Mini-generated errors only. Java errors will be handled in Chapter 6 – I/O.

```
: problem
     "problem" .
     1 throw
     "but you'll never see it again." . ;

: rocket "houston we have a" . problem  "time to abort" . ;
```

```
: launch
     ['] rocket catch
     0= -> cr "Success!" . |
     _  -> cr "Fail!"    . |.
;

launch
houston we have a problem
Fail! ok
```

Exercise 3.9

3.9.1 (*) Test out these examples for yourself. Be sure to be able to understand completely the logical flow, and to be able to explain the outcome at the end.

4 Sequences and Higher Order Functions

Mini supports *lazy sequences* and *higher order functions* (HOFs). Together, these form a powerful abstraction for big data processing. They solve a number of problems:

1. Data processing loops become trivially parallelizable because no mutable state is exposed to the programmer. This is a huge win for computing over distributed datasets.
2. Loops become composable, which can often dramatically simplify programming.

Sequences

A *sequence* is a list (possibly infinite) of items, with a `HEAD` and a `TAIL`. The word `HEAD` accesses the head/first item on the list while `TAIL` accesses the remainder of the list without its head. Any list may be displayed using `.LIST` (of course, an infinite list gives a never-ending listing!)

Sequences can be built “by hand”:

```
nil 0 cons 1 cons 2 cons .list
2 1 0 ok
```

They can be infinite:

```
1 2 ...  .list
1 2 3 4 ( keeps on going! you need to press control-C to stop. )
```

Or, they can be both:

```
1 2 ... "hello" cons .list
hello 1 2 3 4 5 6 7 8 9 ( keeps on going! )
```

Word	Action
HEAD	Puts the first item in the sequence on the stack.
TAIL	Puts the sequence without its head on the stack.
CONS	Adds a head to a sequence. The first argument is the sequence , the second the new head: `( seq x – x:seq )`
.LIST	displays the items in a sequence on screen.
NIL	Puts the empty list on the stack.
NIL?	Tests if the sequence is equal to the special empty list called nil.

EMPTY?	Tests if the sequence is empty.
SEQ?	Tests if the item on the stack is a sequence.
...	Creates an infinite/lazy sequence whose elements are an arithmetic progression defined by the two arguments.
++	Joins together two sequences.

Truncating a Sequence

The word TAKE takes the first *N* elements from any sequence:

```
0 5 ... 10 take .list
0 5 10 15 20 25 30 35 40 45
ok
```

Another way to truncate a list is to use `TAKE-WHILE:`

```
0 5 ... :> 100 < ; take-while .list
0 5 10 15 20 25 30 35 40 45 50 55 60 65 70 75 80 85 90 95
ok
```

`TAKE-WHILE` uses the XT to determine if it should accept further elements from the input list. Both `TAKE` and `TAKE-WHILE` are lazy – they don't perform their job until actually called to (eg, by `.LIST`)

Exercise 4.1

(**) Write a word `..` that takes 2 numbers (start & end) and produces a sequence of numbers from start to end, with skip of 1. Did you consider the case start > end?

Higher Order Functions

A higher order function is a function that transforms, consolidates or combines sequences. These functions are *lazy* in Mini, meaning that they don't immediately perform their task, but wait until an answer is actually needed. This laziness means that these functions can handle infinite lists too, just the same as ordinary ones:

Word	Action
MAP	Takes a sequence and an XT as arguments. The XT must be

(seq xt – seq')	a single-valued function of the form f(x). Mathematically: $map:[x]\rightarrow[f(x)]$
FILTER (seq xt – seq')	Takes a sequence and an XT as arguments. The XT must be a boolean function f(x). The resulting sequence is a sub-sequence of the original, with elements such that f(x) is true: $filter:[x]\rightarrow[x:f(x)=true]$
REDUCE (seq xt – v)	Takes a sequence and an XT as arguments, and runs the XT on each value of the sequence. So, is not lazy, it immediately acts on the input sequence. Therefore, REDUCE may only be run on finite sequences. Typically, in a reduction, you would have an initial value(s) placed on the stack before the sequence (eg, max of a sequence).
ZIPWITH (seq1 seq2 xt -- seq3)	Takes two sequences and an XT as arguments. The XT is used to combine the sequences element-wise, to produce a third sequence: $zipwith:[x][y]\rightarrow[f(x,y)]$

Examples:

Find the numbers divisible by 3, 5 and 7:

```
: div? ( n – xt )
     { n } [: n mod 0 = ;] filter ;

1 2 ... 3 div? 5 div? 7 div? .list
105 210 315 420 525 630 735 840 945 1050 1155 1260 1365  ...
```

Find all squares divisible by 17 and 23:

```
1 2 ... :> dup * ; map 17 div? 23 div? .list
152881 611524 1375929 2446096 3822025 ...
```

Or:

```
: ^2 dup * ;
1 2 ... ' ^2 map 17 div? 23 div? .list
```

Find the sequence resulting from the element-wise multiplication of these sequences:

```
1 2 ... 3 div? 5 div? 7 div?
1 2 ... ' ^2 map 17 div? 23 div?
```

```
' * zipwith .list
16052505 128420040 433417635 1027360320 2006563125 ...
```

In these examples, be careful of overflow. In typical usage of sequences, the sequences are derived from data sources or from live sensors, not analytical sequences like these examples. However, we will later reinstate "big number" capability within Mini (as it was in Mini/Largo).

These examples demonstrate a few things:

1. These examples are traditionally done using *explicit loops* (for-loops, while-loops, etc) in languages like Java, C++, etc.
2. Unlike explicit loops, *implicit loops* are **composable**, meaning that the results of a loop can be fed into a second calculation. This can be used to solve problems which is hard to do with explicit loops.
3. Implicit loops don't need an end condition; this is implicit in the loop itself. For example, we can truncate any one of the lists in the examples using `TAKE`.
4. Less obviously, implicit loops are "stateless" , which goes a long way to ensure that they are easily parallelizable by the runtime. The reverse is true for explicit loops – they are extremely hard to parallelize (why?), and often only under special conditions.

These advantages are especially evident for big data processing:

- Complex analytics are best done by composing simpler calculations,
- The data sets are of indefinite size, and may vary over time,
- The datasets are often distributed. Parallelization of the calculation is often necessary.

Exercise 4.2

4.2.1 (*) Try out the examples and be sure you understand them thoroughly.
4.2.2 (**) Using explicit loops, write pseudocode to find the first 100 numbers in the third example.
4.2.3 (**) `ZIP` is a frequently used word that combines two lists into a list of pairs: $zip : [x][y] \rightarrow [(x, y)]$ Write `ZIP`. (Hint: use the appendix to find out how to construct pairs).

Reductions

`MAP`, `FILTER` and `ZIPWITH` all transform sequences into new sequences, lazily. Unlike its cousins, `REDUCE` applies the XT to each value fo the sequence, (but without building a new sequence like `MAP`), and it is not lazy. Since `REDUCE` will try to iterate over the entire sequence, so you must ensure it is finite before attempting a reduction.

Examples:

```
0 1 2 ...  100 take ' max reduce .
100.0 ok
```

In this example, the word MAX needs two arguments. One is already on the stack (0) and is constantly updated by MAX during the reduction. The second is provided by the sequence as REDUCE iterates through it.

Exercise 4.3

4.3.1 (*) Write .LIST in terms of REDUCE.

4.3.2 (*) Write a word that sums all values in a finite sequence.

4.3.3 (**) Write a word to find the minimum of a finite sequence without making any assumptions about the upper bound. Does your word work for empty sequences?

4.3.4 (**) Write a word, REVERSE that reverses any finite list. Does your word work for empty sequences?

Recursive Sequences

Many sequences are conveniently expressed in a recursive manner. For example, the fibonacci series is: $f(n)=1,2...f_{(n-1)}+f_{(n-2)}...$ A recursive sequence is defined using 2 words:

Word	Action
{:	called DELAY-SEQ, puts a temporary pointer to the sequence to be defined on the stack.
;}	called END-DELAY-SEQ completes the delayed sequence. It takes as its first argument the original delay, and as it second argument the sequence body, which is recursively defined.

Examples:

We want to generate the sequence (1,0,1,0,...) This sequence is obviously of the form $\alpha=(1,0,\alpha)$ We use this recursive definition to create the sequence:

```
{:     \ this is          α
dup 0 cons 1 cons   \ this is    (1,0,α)
;}  \ this step completes    α=(1,0,α)
```

```
10 take .list
1 0 1 0 1 0 1 0 1 0
ok
```

As a second example, consider the Fibonacci series. This series can be expressed as $\alpha=(0,1,\alpha)+tail(0,1,\alpha)$ Where "+" is element-wise addition. We therefore define:

```
{:
   dup 1 cons 0 cons
   dup tail
     ' + zipwith
;}
10 take .list
1 2 3 5 8 13 21 34 55 89
ok
```

To understand how this works, you need to unravel the recursive sequence step by step:

Step	Expression	α
0	(0, 1, α) + tail(0, 1, α) = (0, 1, α) + (1, α) = [0+1] : (1, α) + α = 1 : (1, α) + α	(1,...)
1	since α = head(α): tail(α) = 1:tail(α) = 1 : (1, 1, tail(α)) + (1, tail(α)) = 1 : 2 : (1, tail(α)) + tail(α)	(1,2,...)
2	since tail(α) = head tail(α): tail tail(α) = 2:tail tail (α) = 1 : 2 : (1, 2, tailtail(α)) + (2, tailtail(α)) = 1 : 2 : 3 : (2, tailtail(α)) + tailtail(α)	(1,2,3,...)
3	since tailtail(α) = head tailtail(α): tail tailtail(α) = 3:tail tailtail(α) = 1 : 2 : 3: (2,3,tailtailtail(α)) + (3, tailtailtail(α)) = 1 : 2 : 3 : 5: (3, tailtailtail(α)) + tailtailtail(α)	(1,2,3,5,...)

And so on.

Exercise 4.4

4.4.1 (**) Is the recursive Fibonacci sequence a *recursive* algorithm or *iterative*? Defend your answer with facts.

4.4.2 (**) Given any input finite sequence write a word `REPEAT` that creates a new sequence which is an unending repetition of the input.

4.4.3 (**) Given a sequence $s=(s_0, s_1, s_2, ...)$ write a word `CUMULANT` that gives its cumulant $c(s)=(s_0, s_0+s_1, s_0+s_1+s_2, ...)$

4.4.4 (**) Write the word `...` in terms of `CUMULANT` and `REPEAT`

4.4.5 (**) Write the word `ACCUMULATE` as a generalization of `CUMULANT` , where the initial value and combination operator are passed as parameters. For example, you should be able to redefine `CUMULANT` as:

```
: CUMULANT ( seq — 'seq )
          0 ['] + ACCUMULATE ;
```

4.4.6 (**) Given any sequence, s write the word `TAILS` that is a sequence $tails: s \rightarrow (s, tail(s), tail^2(s), ...)$

Exercise 4.5

4.5.1 (**) Write a module called `*SEQMATH` that defines `*, -, +` and `/` between two sequences. Put `REPEAT` and `CUMULANT` into this module.

4.5.2 (*) Add `MIN, MAX` for sequences to your module.

4.5.3 (*) Add sequence extensions for all the other words in the Math module (see Appendix) to your module. Can `TAN` be defined in terms of `COS`, `SIN` and `/` ?

4.5.4 (**) Add the word `*c` that multiplies a sequence with a number. Make this word "smart" so that it will work regardless of the relative positions of constant & seq: `*c: ( seq c | c seq — 'seq )`.

4.5.5 (**) Use your module functions to find the moving average of size 3 of a list $avg3(s)=(\frac{s_0+s_1+s_2}{3}, \frac{s_1+s_2+s_3}{3}, ...)$,given any list $s=(s_0, s_1, s_2, ...)$

4.5.6 (***) Generalize `AVG3` to `MAVG` , which is able to average any length window N, which is given as input along with the initial sequence: `MAVG: ( seq N — 'seq )` Put `MAVG` into your module.

Data Sources

So far, we have dealt with analytical number sequences. The word **`SEQ`** converts an XT into a sequence. This is especially useful when the XT is "stateful" -- ie, it represents:

- data read from a file or database. This could be text, numbers, etc.
- sensor data over the network or file,
- data generated using stateful words.

Example: Linear Congruential Random Number Generator

As an example, consider generating a sequence of random numbers using LCG (https://en.wikipedia.org/wiki/Linear_congruential_generator) These are poor random number generators, but are trivial to implement.

If we had a word `LCG ( s – xt )` that created an XT from a seed **s**, then `LCG SEQ` would represent a sequence of pseudorandom numbers.

Exercise 4.6

4.6.1 (**) Give a working implementation of an LCG. Use the Wikipedia link and the "Numerical Recipes" recommended settings.

4.6.2 (***) Write a word that transforms a text file into a sequence, reading line-by-line. Hint: Use the Appendix.

5 Macros

What are Macros?

Macros are words that build other words. They are helpful in the following contexts:

- **Building new control words:** Mini has words to allow programmers build new control structures that simplify particular programming problems or implement a programming idiom. A disciplined understanding of macros – borrowed from lisp -- helps us take this to new levels.
- **Avoiding repetitive blocks of code:** Macros often encapsulate reusable idioms that reduce code duplication.
- **Making code more efficient**: Sometimes, the value of a variable is only known at runtime. Macros can be used to create code blocks at runtime, or optimize code at runtime. Example: Just-in-time compilation.
- **Help build domain-specific languages:** Macros allow us to build meta-words (words that create other words) to solve complex programming problems.

Besides the `CREATE` and `DOES>` words that are often indispensable in macros, Mini extends macro writing with the notion of nestable quotations, lexical closures, lambdas and references. We will meet these next.

Nestable Quotations

Mini has *nestable quotations* – these are the words [: and ;] for example:

```
: hw  [: “Hello World” . cr ;] ;

hw .S
<1> com.terraweather.mini.XT@62c09554
ok

execute
Hello World
ok
```

Lexical Closures

Quotations allow us to create *lexical closures* -- closures in which specific variable bindings apply throughout that closure *and* quotations nested within

it. For example:

```
: hw2 { x } [:  x 1+ . ;]  ;
```

Here, ***x*** is a local variable of hw2:

```
see hw2
[0]  write local: >x<
[1]  com.terraweather.mini.core.Block@518bf072
[2]  read local: >x<
[3]  1+
[4]  .
[5]  EndBlock<com.terraweather.mini.core.Block@518bf072>
ok
```

But ***x*** is inlined into the closure when hw2 is run:

```
41 hw2 decompile
[0]  Literal<41>
[1]  1+
[2]  .
ok
```

The decompilation clearly shows that the lexical variable “x” has been replaced with the value 41 at runtime. A similar inlining occurs for closures nested within the enclosing closure:

```
: hours-to-millis 60 * 60 * 1000 * ;

: hw3 ( timezone-hours -- local-timestampXT )
     { tz } [:
          tz hours-to-millis  now + { t } [:
               “I was created at: ” . t . “utc+” . tz .
          ;]
     ;]
;

8 hw3 dup decompile
[0]  Literal<8>
[1]  HOURS-TO-MILLIS
[2]  com.terraweather.mini.time.time$1@54281d4b
[3]  com.terraweather.mini.number.number$8@159b5217
[4]  write local: >t<
[5]  com.terraweather.mini.core.Block@19e3118a
[6]  Literal<I was created at: >
[7]  .
[8]  read local: >t<
[9]  .
```

```
[10] Literal<utc+>
[11] .
[12] Literal<8>
[13] .
[14] EndBlock<com.terraweather.mini.core.Block@19e3118a>
ok

execute decompile
[0]  Literal<I was created at: >
[1]  .
[2]  Literal<1460457021180>
[3]  .
[4]  Literal<utc+>
[5]  .
[6]  Literal<8>
[7]  .
ok
```

Read Time, Run Time & Compile Time

For any Mini word:

Read Time refers to the time when the textual source code for that word is read in, parsed and converted into executable code. Read Time occurs only once.

Run Time refers to the time when the executable code for that word is run. Since a word could be run multiple times, Run Time could occur more than once.

Compile Time refers to any time when executable code is created. This could occur towards the end of Read Time (which is usual in traditional Forth), or at Run Time (when a macro is run or when a closure is being created).

For closures, Mini executes “immediate” words , except for [and] (CLOSE/OPEN BRACKET) during Read Time only. [ ] themselves are run only during the Compile Time for that closure.
Example:

```
: say-hi “Hello!” . cr ; immediate

: test
     [:   \ outer closure

          say-hi \ this is fired immediately at TEST's Read Time
```

```
          [ “How are you?” . cr ] \ this is fired at the outer
                                  \ closure's Compile time.
                                 \ which is TEST's Run Time.

          [:  \ inner closure

               say-hi \ this is fired immediately
                      \ at TEST's Read Time

               [ “I am fine” . cr ] \ this is fired at the inner
                                    \ closure's Compile time.
                                    \ which is the outer closure's
                                    \ Run Time.
          ;]
     ;]
;
```

NIF, Macro

As an example of a simple macro, consider the word NIF, which compiles the appropriate XT ('p, 'z or 'n) into the enclosing closure depending on the value on the stack:

```
\ ( expr 'p 'z 'n -- )
: nif,
     { 'p  'z  'n }
     dup 0>  ->  drop 'p  |
         0<  ->  'n       |
     otherwise   'z       |.
     compile, ;

: too-hot "too hot!" . ;
: too-cold "too cold!" . ;
: just-nice "just nice ... " . ;

: goldilocks ( tempC — xt )
     26 - { t }
     [:
          [  t
             ['] too-hot
             ['] just-nice
             ['] too-cold
                    nif, ]
     ;]
;
```

Note that in Mini, immediate words will run immediately within a closure definition regardless of nesting level, *except* for compilation state-change

words [and] will run only when their enclosing context is being compiled. This is the reason to use ['] within [...] in the example above – we continue to be in compile state even after [is encountered because it is within a quotation, and ['] is therefore the right tick to use. ['] is immediate, and sso will execute during the compilation of GOLDILOCKS.

Running the example:

```
26 goldilocks decompile
[0]  JUST-NICE
ok
```

Lambdas

In the preceding example, a lexical closure permanently binds a given name to a value. It is also possible to bind variables on the fly:

```
: repeater  [: { x } "You said" . x . "!"  . cr ;] ;
ok

"hey there!" repeater execute
You said hey there! !
ok
```

Whenever a lexical variable name clashes with that of a local one, the lexical variable wins:

```
: hw5 { x } [:  { x } x . ;] ;
ok

see hw5
[0]  write local: >x<  ---- this is a lexical variable for the
closure, but local variable for the word HW5.

[1]  com.terraweather.mini.core.Block@792442a2

[2]  write local: >x<  ---- this is a local variable for the
closure.

[3]  read local: >x<
[4]  .
[5]  EndBlock<com.terraweather.mini.core.Block@792442a2>
ok

32 hw5 decompile
[0]  write local: >x<
```

```
[1]  Literal<32>
[2]  .
ok
```

The binding of the outer lexical variable x => 32 is **always** used first when the closure is generated. The local variable, x is bound, but never accessed within the closure. This can be a source of subtle bugs. Fortunately, the scope of any variable (lexical or local) is limited to the enclosing word or quotation.

References

Lexical variables are frozen during compilation of the closure and local variables are always reset across multiple invocations of a closure. So how to save state? The answer is to use *references*, which can be constructed from closures:

```
: ref ( x -- r ) [: [ , ] ;]  ;
```

There are other ways to create references, but this is a particularly convenient one. Consider this example:

```
:  elephant
     ref  { x } [:  x @ 1+ dup x ! “Number is now:” . . cr ;]  ;


41 elephant

dup execute
Number is now: 42
ok
dup execute
Number is now: 43
ok
```

In object-oriented speak, `ELEPHANT` produces a simple “*object*” (ie, code with mutable state), with just one implicit function, (viz, to increment and display the “number” internal state).